Scalable Iterative Graph Duplicate Detection

Melanie Herschel, Felix Naumann, Sascha Szott, Maik Taubert

Scalable Iterative Graph Duplicate Detection

Melanie Herschel, Felix Naumann, Sascha Szott, and Maik Taubert

Abstract—Duplicate detection determines *different* representations of real-world objects in a database. Recent research has considered the use of relationships among object representations to improve duplicate detection. In the general case where relationships form a graph, research has mainly focused on duplicate detection quality/effectiveness. Scalability has been neglected so far, even though it is crucial for large real-world duplicate detection tasks. We scale-up duplicate detection in graph data (DDG) to large amounts of data and pairwise comparisons, using the support of a relational database management system. To this end, we first present a framework that generalizes the DDG process. We then present algorithms to scale DDG in space (amount of data processed with bounded main memory) and in time. Finally, we extend our framework to allow batched and parallel DDG, thus further improving efficiency. Experiments on data of up to two orders of magnitude larger than data considered so far in DDG show that our methods achieve the goal of scaling DDG to large volumes of data.

Index Terms—Duplicate detection, data cleaning, data integration, record linkage, entity resolution, scalability, parallelization.

1 Introduction

Duplicate detection algorithms (surveyed in [1]) determine which different entries in a database in fact represent the same real-world object. They are for instance used in data cleaning [2] and integration [3] and have been studied extensively for data stored in a single relational table.

Recently, a new class of duplicate detection algorithms emerged that considers the more complex structure of data (more than one table, foreign keys). We refer to these algorithms as *duplicate detection in graph data*, DDG for short. These algorithms detect duplicates between object representations, so called *candidates*, by utilizing relationships between candidates to improve effectiveness. Within this class, we focus on those algorithms that iteratively detect duplicates when relationships form a graph [4], [5], [6]. In this paper, reference to DDG algorithms implies the restriction to these iterative algorithms. To the best of our knowledge, none of the proposed algorithms within this class has explicitly considered scalability yet. This paper aims at filling this gap and extends on previous work [7]. Before outlining our contributions, we present an example that illustrates DDG and highlights its benefits.

The tables in Fig. 1 describe movies, actors, and which actor starred in which movie. It is obvious to humans that all three movies represent a single movie and that there are only three distinct actors, despite some typos.

- *M. Herschel is with the Wilhelm-Schickard Institut für Informatik, Universität Tübingen, Sand 13, 72076 Tübingen, Germany. E-mail: melanie.herschel@uni-tuebingen.de.*
- *F. Naumann is with the Hasso-Plattner-Institut für Softwaresystemtechnik, Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany. E-mail: naumann@hpi.uni-potsdam.de.*
- *S. Szott is with the Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr. 7, 14195 Berlin, Germany. E-mail: szott@zib.de.*
- *M. Taubert is with Biotronik SE & Co. KG, Woermannkehre 1, 12359 Berlin, Germany. E-mail: maik.taubert@biotronik.de.*

Manuscript received 30 Jan. 2010; revised 10 Sept. 2010; accepted 29 Mar. 2011; published online 27 Apr. 2011.
Recommended for acceptance by W.-S. Han.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-01-0058.
Digital Object Identifier no. 10.1109/TKDE.2011.99.

Example 1. We consider movies, titles, and actors as candidates, which we identify by an ID with prefix m, t, and a, respectively, for future reference. We use the prime notation (') to indicate duplicates. Title candidates illustrate that candidates do not necessarily correspond to a complete relation.

Initialization. DDG algorithms represent data as a graph. Fig. 2 depicts a possible graph for our movie scenario. The graph includes one candidate node per candidate, whose descriptive information, given by the attributes of the relational table, is represented as attribute nodes associated with the corresponding candidate node. Candidate nodes are connected by directed edges, called dependency edges. Intuitively, these edges represent the fact that finding duplicates of the target candidate depends on finding duplicates of the source candidate. The latter are called influencing candidates.

Next, DDG initializes a priority queue, PQ, of candidate pairs. Each pair in PQ is compared based on some similarity measure that considers both neighboring attribute nodes and influencing candidate nodes. If the similarity is above a given threshold, the pair is classified as a duplicate. Let the initial PQ be $\langle (m1, m1'), (m1, m1''), (m1', m1''), (t1, t2), (t1, t3), (t2, t3), (a1, a2), \ldots \rangle$.

Iterative phase. We now start comparing pairs in PQ. The first pair retrieved from PQ is $(m1, m1')$, which is classified as nonduplicate, because the movies' sets of influencing neighbors (the respective actor neighbors) appear to be nonduplicates. The same occurs when subsequently iterating through movie pairs and title pairs. When comparing $(a1, a1')$, we finally find a duplicate, because the actor names are similar, and there are no influencing neighbors. Having found this duplicate potentially increases the similarity of $(m1, m1')$, because the movies now share common actors. Hence, $(m1, m1')$ is added back to PQ for further inspection. The same occurs in the subsequent iterations where duplicate actors are detected. Now, when we compare movies again they can be classified as duplicates, because they share a significant amount of influencing actor neighbors. In the end, we identify all duplicates correctly.

MOVIE

MID	Title
m1	Troy
m1'	Troja
m1"	Illiad Project

ACTOR

AID	Name
a1	Brad Pitt
a2	Eric Bana
a1'	Brad Pit
a2'	Erik Bana
a3	Brian Cox
a1"	Prad Pitt
a3'	Brian Cox

STARS_IN

AID	MID
a1	m1
a2	m1
a1'	m1'
a2'	m1'
a3	m1'
a1"	m1"
a3'	m1"

Fig. 1. Sample movie database.

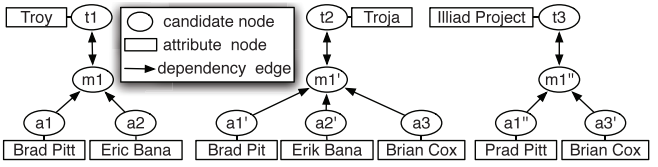


Fig. 2. Sample reference graph.

Recomparisons, e.g., of the movie candidates, increase effectiveness and should thus be performed. However, had we started by comparing actors, we would have avoided classifying movies a second time and still have obtained the same result. Clearly, comparison order affects efficiency.

Contributions. Whereas effectiveness and priority queue order maintenance for efficiency have been considered in the past, scalability of DDG has not been addressed so far. This paper aims at filling this gap. More precisely, our contributions are: 1) A general framework that suits several iterative DDG algorithms; 2) algorithms to scale up the general DDG process described by our framework so that DDG algorithms can be applied to large amounts of data; and 3) further efficiency improvements through parallelization and batched processing.

Structure. We cover related work in Section 2. We then present our framework for iterative DDG in Section 3. Section 4 presents how we scale-up DDG. Our extension to parallel and batched execution of DDG, called PDDG, is discussed in Section 5. We cover our evaluation in Section 6 before the conclusion in Section 7.

2 Related Work

We discuss related work on the two main topics pertinent to this work: DDG and parallelization.

Duplicate detection in graph data. We distinguish between three approaches used for DDG: 1) machine *learning*, where models and similarity measures are learned [8], [9], 2) the use of *clustering* techniques [10], [11], [12], [13], and 3) *iterative* algorithms that classify one pair of candidates at every iteration [4], [5], [6]. We further discern whether an algorithm mainly focuses on effectiveness, efficiency, or scalability. Research on effectiveness is concerned with improving precision and recall, which is a goal common to all DDG algorithms. Research on efficiency aims at improving the runtime [5], [10], [12], [13], [14]. To apply methods to very large data sets, it is essential to scale not only in time but also to scale in space, for which RDBMSs are commonly used for duplicate detection in general [15], [16], [17], but which has not been explored for graph duplicate detection. Let us now take a closer look at algorithms of the class we consider, i.e., iterative DDG.

Dong et al. [4] perform duplicate detection in a scenario where relationships between persons, publications, etc., form a graph. At each iteration, the first pair in the priority queue is retrieved, compared, and classified as nonduplicate or duplicate. In the latter case, relationships are used to propagate the decision to neighbors, which potentially become more similar due to the found duplicate. To reflect this propagation in the priority queue, pairs that potentially are more similar are either put at the head of the priority queue or in the tail, depending on the type of relationship. Also, the constraint that two candidates are nonduplicates is explicitly considered by this approach. We do not consider such negative evidence in this paper but our framework can be extended to incorporate it. Efficiency and scalability are not considered by their approach.

In studying the impact of comparison order on DDG efficiency [5] we show that a poor choice of comparison order can compromise efficiency due to a large number of recomparisons in data graphs with high connectivity and present a heuristic to determine an efficient comparison order.

The RC-ER algorithm [6] reevaluates similarities of candidate pairs at each iterative step, and selects the most similar pair at each iteration. Duplicates are merged together before the next iteration, so effectively clusters of candidates are compared. This merge necessitates updates in the reference graph and the priority queue. The authors describe the use of a binary-heap in main memory to efficiently maintain the order of the priority queue. However, scalability is not discussed.

Parallelization. By parallelization, we understand the distribution of an algorithm such that it can be computed simultaneously on one or multiple computing units [18]. To the best of our knowledge, no work has yet considered parallelization of general DDG.

In [19] an approach for parallelizing duplicate detection on a single relational table is presented. That work does not address issues of graph data. The parallel version of Swoosh [14] processes two related candidate types in parallel and propagates duplicates among the two processes. However, as opposed to what all other DDG algorithms ensure, the propagation is not exclusively targeted toward candidate pairs whose similarity potentially increases, so pairs whose similarity is guaranteed to not change may be recompared as well. We therefore do not view this approach as parallelization of general DDG.

Due to interdependent classifications, the MapReduce framework [20] is not directly applicable to the general parallelization of DDG. Therefore, we devise a specialized framework, named PDDG, that builds upon the nonparallel DDG framework discussed next.

3 The DDG Framework

The methods we propose to scale-up DDG are intended to be as general as possible to fit many algorithms. Therefore, we first generalize existing methods into a unified framework.

3.1 Definitions for Unified DDG

We reuse and extend several definitions we have defined for duplicate detection in tree data [21], starting with candidates.

Definition 1 (Candidates). *Let $C_T = \{c_1, c_2, \ldots, c_k\}$ be the set of candidates of a given object type T. To denote the type of a*

particular candidate c, we use T_c. The set of all candidates, no matter their type, is denoted as C.

During the iterative phase of DDG, candidate pairs are classified based on a similarity measure.

Definition 2 (Duplicate classification). *Given a similarity measure sim and a threshold θ, we classify c and c' as duplicates if $sim(c, c') > \theta$, and as nonduplicates otherwise.*

Intuitively, sim considers both object descriptions and influencing candidate pairs in its computation.

Definition 3 (Object description). *Let $\mathcal{A}_T$ be a set of attribute names for candidates of type T. Then, $\mathrm{OD}_\alpha(c)$ defines the attribute values of an attribute named $\alpha \in \mathcal{A}_T$ that are part of c's* object description *OD. The complete OD of a candidate $c \in C_T$ is given by $OD(c) = \{(\alpha, v) | \alpha \in \mathcal{A}_T \wedge v \in OD_\alpha(c)\}$. When the attribute name α is clear from the context, we write (v) instead of (α, v) for brevity.*

Example 2. A candidate of type actor is described by only its name, hence $\mathrm{OD}(\mathrm{a1}) = \mathrm{OD}_{\mathrm{Name}}(\mathrm{a1}) = \{(\mathrm{BradPitt})\}$.

Definition 4 (Influencing & dependent candidates). *The set of* influencing candidates *$I(c)$ of a candidate c is a set of candidates different from c whose similarity (or duplicate status) to other candidates affects the similarity of c to other candidates (see Definition 5 for the definition of the other candidates). Analogously,* dependent candidates *of c are candidates different from c whose similarity is influenced by c, i.e., $D(c) = \{c' | c \in I(c')\}$.*

Placing candidates into $I(c)$ and $D(c)$ is for instance based on foreign key constraints or domain knowledge provided by a domain expert. It reflects the graph structure of the data.

Example 3. Fig. 2 shows $I(\mathrm{m1}) = \{\mathrm{a1}, \mathrm{a2}, \mathrm{t1}\}$ and $D(\mathrm{m1}) = \{\mathrm{t1}\}$. Similarly, $I(\mathrm{m1''}) = \{\mathrm{a1''}, \mathrm{a3'}, \mathrm{t3}\}$ and $D(\mathrm{m1''}) = \{\mathrm{t3}\}$.

Influencing and dependent candidate pairs of two candidates are obtained by forming the cross product between their respective influencing and dependent candidates of same type.

Definition 5 (Influencing & dependent candidate pairs). *The* influencing candidate pairs *of some candidate pair (c, c') are defined as $I(c, c') = \{(i, i') \in I(c) \times I(c') | T_i = T_{i'}\}$. Analogously, the* dependent candidate pairs *of (c, c') are $D(c, c') = \{(i, i') \in D(c) \times D(c') | T_i = T_{i'}\}$.*

Example 4. Fig. 2 shows $I(\mathrm{m1}, \mathrm{m1''}) = \{(\mathrm{a1}, \mathrm{a1''}), (\mathrm{a1}, \mathrm{a3'}), (\mathrm{a2}, \mathrm{a1''}), (\mathrm{a2}, \mathrm{a3'}), (\mathrm{t1}, \mathrm{t3})\}$, and $D(\mathrm{m1}, \mathrm{m1''}) = \{(\mathrm{t1}, \mathrm{t3})\}$.

3.2 Components of Unified DDG

We observe that algorithms for iterative DDG have in common that 1) they consider data as a graph, 2) they perform some preprocessing before candidates are compared to obtain initial similarities and avoid recurrent computations, and 3) they compare pairs of candidates iteratively in an order that possibly changes at every iteration where a duplicate is found, requiring the maintenance of a priority queue. Merging or enriching detected duplicates is also a common technique to increase effectiveness and requires updating the graph. Based on these observations, we devise the general DDG framework illustrated in Fig. 3.

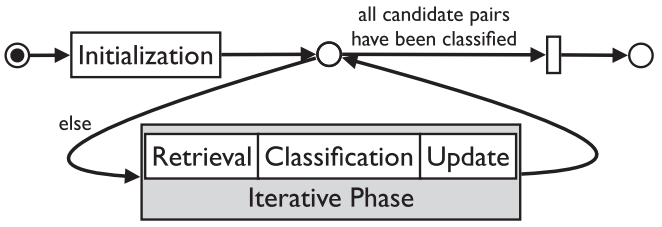


Fig. 3. General DDG workflow.

Unified graph model. Dong et al. [4] use a *dependency graph* whose nodes are pairs of candidates $\rho_c = (c_1, c_2)$ or pairs of attribute values $\rho_\alpha = (v_1, v_2)$, and an edge between ρ_c and ρ_α exists if v_1 is an attribute value of c_1 and v_2 is an attribute value of c_2. Further edges between candidate pair nodes ρ_{c_1} and ρ_{c_2} exist if ρ_{c_1} is an influencing or dependent pair of ρ_{c_2}. Opposed to that, Bhattacharya and Getoor [22] define a *reference graph*, where nodes do not represent pairs, but candidates, and edges relate influencing or dependent candidates. Other DDG algorithms use graphs that basically fall into one of these two categories: nodes either represent single candidates [5], [10], or pairs [8]. Attribute values are usually treated separately. In the context of duplicate detection, the dependency graph can be derived from the reference graph. In general, a dependency graph is more expressive, but its full expressiveness is not used in DDG. Hence, our unified graph model for DDG is a reference graph. Fig. 2 is a reference graph for our movie example.

Definition 6 (Reference graph). *A reference graph $G = (\mathcal{V}_C, \mathcal{V}_\mathcal{A}, \mathcal{E})$ consists of a set of candidate nodes $\mathcal{V}_C$, a set of attribute nodes $\mathcal{V}_\mathcal{A}$, and a set of dependency edges $\mathcal{E}$. A candidate node $V_c \in \mathcal{V}_C$ exists for every candidate $c \in C$. Every element of an OD is represented by an attribute node $V_\alpha \in \mathcal{V}_\mathcal{A}$. A dependency edge $(V_c, V_{c'}) \in \mathcal{E}$ is directed from V_c to $V_{c'}$ if $c' \in D(c)$.*

Unified DDG initialization. To detect duplicates, DDG algorithms set up a *priority queue PQ* where the priority of a candidate pair is computed according to an algorithm-specific method (see Section 2 for different strategies). Theoretically, all pairs of candidates in G of equal type are added to PQ. However, blocking [23] or filtering [15] are commonly used to significantly reduce the number of pairs entering PQ, thus avoiding the issue of an otherwise quadratic blowup. Also, the set of pairs in PQ may change during the iterative phase, which is especially useful when duplicates get merged.

Also as part of initialization, DDG algorithms have a jump-start phase, where, e.g., attribute similarities are precomputed for later use during candidate classification [6] or duplicates that can be identified with a less complex measure are detected and not further compared during DDG [4]. Essentially, the jump-start phase performs precomputations used later on.

Unified iterative phase. After initialization, we classify candidate pairs as duplicates or nonduplicates in the

iterative phase. Existing algorithms maintain PQ in main memory. At every iteration step, the first pair in PQ is *retrieved*, then *classified* using a similarity measure, and finally the classification causes some *update* in PQ or the graph (adding pairs to PQ, enrichment in [4], duplicate merging and similarity recomputation in [22]) before the next iteration starts.

In general, PQ has to be reordered whenever a duplicate is detected to reduce the number of recomparisons, for which [4], [5], [22] devise different strategies. Reordering is an expensive task for large priority queues. However, by maintaining the order in PQ, recomparisons are potentially avoided, as we illustrated in Example 1. This in turn may positively affect runtime. For instance, on graphs with high connectivity we observed runtime savings of up to 40 percent [24].

4 SCALING-UP DDG

We scale-up DDG to large amounts of data using the support of an RDBMS, because it provides fast implementations of crucial operations, such as sorting large amounts of data. In addition, this choice allows easy deployment of our methods to real-world scenarios where data are stored in relational databases, as we did in [25]. Finally, using an RDBMS follows the spirit of prior work on duplicate detection scalability [15], [16], [17].

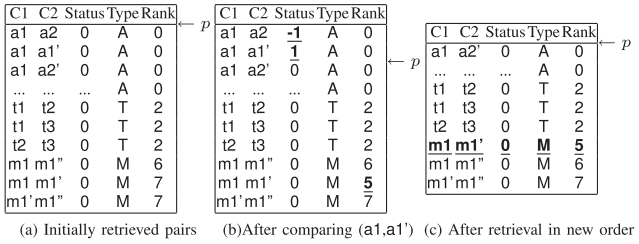
4.1 Scaling-up Initialization

First, let us briefly summarize how we scale initialization (see [26] for details). Essentially, we create the following tables in a database to store the reference graph structure and precomputed values to speed up similarity computation.

- $EDGES(S, T, Weight, Type)$,
- $PQT(C1, C2, Status, Type, Rank)$,
- $ODWeight\_T_i\_\alpha_j(v, Weight)$,
- $ODSim\_T_i\_\alpha_j(v_1, v_2, Sim)$,
- $DEP(S1, S2, T1, T2, Status\_S, Status\_T, Weight\_S, Weight\_T)$.

Assuming that candidates are already stored in a database we reflect the reference graph by storing edges as source and target candidate ids, and the type of an edge (required by some algorithms, e.g., [4]) in *EDGES*. Pairs in PQ are added to the *PQT* relation together with their duplicate status (e.g., duplicate, nonduplicate), their candidate type, and their rank within PQ. Tables *ODWeight* and *ODSim* store for each OD attribute α_j of each candidate type T_i weights of individual values and pairwise value similarities, respectively. Finally, *DEP* materializes the relationship of influencing candidate pairs (a source pair and a target pair) together with the pairs' status and weight. This avoids expensive and long running join computations on edge tuples during the iterative phase. In our implementation, we do not use filtering and we insert all possible pairs into *PQT*. We use IDF as weight function and precompute edit-distance-based similarity.

Loading the data into the tables described above may be a time consuming task when performing it for the first time. In practice, however, we observe that this process runs within hours for millions of candidates and performing precomputations significantly improves the runtime of the iterative phase, despite the overhead caused by looking up these values in the database. Also, the precomputed tables may be updated incrementally so subsequent DDG runs require less time for initialization.



(a) Initially retrieved pairs (b)After comparing (a1,a1') (c) After retrieval in new order

Fig. 4. *PQT* states using RECUS/DUP on movie example.

We now discuss how we scale up the iterative part of DDG. We first focus on scaling-up retrieval and update, and postpone the discussion of classification, for which we present solutions orthogonal to the choice of the retrieval and update algorithm.

4.2 Scaling-up Retrieval and Update

A straightforward approach to scale-up DDG is to map the DDG process (Fig. 3) from main memory to a database. We refer to this baseline algorithm as RECUS/DUP, as the **Re**trieval-**C**lassify-**U**pdate-**S**ort process is guided by **dup**licate classifications. Although straightforward, we discuss RECUS/DUP as the basis for our improved algorithm.

Let us use a heuristic approach that sorts pairs (c, c') in *PQT* in ascending order of a rank computed as

$$|I(c)| + |I(c')| - 2 \cdot |N_{ip}^{\approx}(c, c')|,$$

where $N_{ip}^{\approx}(c, c')$ is the set of currently known duplicate pairs among $I(c)$ and $I(c')$ (see Definition 7). Intuitively, the larger the number of influencing candidates that are not duplicates of each other, the more classifications may trigger a recomparison of pair (c, c'). To avoid these, the comparison of (c, c') should be performed late in the process. This ranking corresponds to one heuristic explored in [5]. Although not the best in practice, its simplicity is suited for illustration. Other heuristics include adding pairs to the head or tail of the priority queue [4] in which case the new rank should be chosen as the minimum or maximum possible rank; or the priority queue is sorted according to the actual distance (similarity) measure [6], [22], in which case the rank is equal to the (inverse) similarity.

RECUS/DUP. Candidate pairs are retrieved from the database as long as *PQT* contains pairs with an unknown duplicate status (identified by 0). To retrieve pairs, we issue the SQL query **`SELECT * FROM PQT WHERE STATUS = 0 ORDER BY RANK`** and iterate over returned tuples. The initial result retrieved by RECUS/DUP is depicted in Fig. 4a. Pointer p is used to iterate over returned tuples. As long as no duplicate is detected during classification, we simply advance p after updating the status of the current pair to -1, which identifies a nonduplicate.

When a duplicate is detected, we need to not only update the pair's status to 1 to mark it as a duplicate, but we have to also update the rank of dependent neighbor pairs. Fig. 4b shows the processing status right after duplicate $(\mathrm{a1}, \mathrm{a1}')$ has

been found. We observe that this classification requires a recomputation of the rank of the dependent pair $(\mathrm{m1}, \mathrm{m1}')$, which changes from 7 to 5. Hence, the order of the initially retrieved pairs no longer corresponds to the order dictated by the rank. Therefore, we terminate the iteration over the retrieved pairs, perform all necessary updates and issue the SQL query described above over the updated *PQT*. Necessary updates may include updating the rank of all dependent pairs, which are efficiently retrieved using the *DEP* relation and updating their status to 0 if necessary; duplicate merging or enrichment, implemented through updates in *EDGES* and OD attributes; or any other operation expressible by an appropriate function that updates graph tables and precomputed tables as required by the operation. Fig. 4c shows the pairs retrieved by the second call of the SQL query. Pointer p is reset to point to the first pair. The subsequent processing is analogous to the procedure described above. Obviously, RECUS/DUP compares all pairs in the correct order w.r.t. the ranking heuristic.

RECUS/DUP makes the least possible use of main memory by keeping a single pair and possibly a dependent pair in main memory at every iteration, plus some information to compute similarity. However, as experiments show (Section 6), sorting *PQT* is very time consuming. The algorithm described next, RECUS/BUFF, reduces the sorting effort by using main memory more extensively, but with an upper bound, while still guaranteeing that pairs are processed in the correct order.

RECUS/BUFF uses an in-memory buffer B of fixed size to avoid sorting *PQT* each time a duplicate is found. The intuition is that although ranks of several pairs may change after a duplicate is detected, sorting *PQT* immediately after finding the duplicate is not always necessary. For instance, in our example of Fig. 4, although the rank of (m1, m1') has changed from 7 to 5, all uncompared actor and title pairs are still in front of (m1, m1') so the order of pairs in the head of the sequence of retrieved pairs remains the same. Hence, sorting the priority queue does not immediately affect the comparison order and can be avoided. To this end, we use buffer B to temporarily store pairs with their new rank and status. B maintains the pairs in ascending order of their rank (the same order as used for *PQT*). By adequately adapting the retrieval and update phase from RECUS/DUP this technique avoids sorting the much larger *PQT* table, while comparing the pairs in the same order as RECUS/DUP.

Fig. 5 illustrates how RECUS/BUFF behaves. Initially, we issue the same query as for RECUS/DUP and B is empty (see Fig. 5a). In addition to a pointer p over pairs from the database, we have a pointer p_B to iterate over pairs in B.

RECUS/BUFF performs like RECUS/DUP as long as it does not detect a duplicate. When a duplicate is detected, its status is updated to 1 and dependent pairs are retrieved and their new rank is computed. Opposed to RECUS/DUP, RECUS/BUFF does not update *PQT* to reflect the new ranks of dependent pairs. Instead, these pairs are added to B in the order of their new rank as long as the buffer does not overflow. The status of these pairs is set to 0. Fig. 5b shows the content of B after duplicate (a1, a1') has been found. Note that p has moved forward to the next pair.

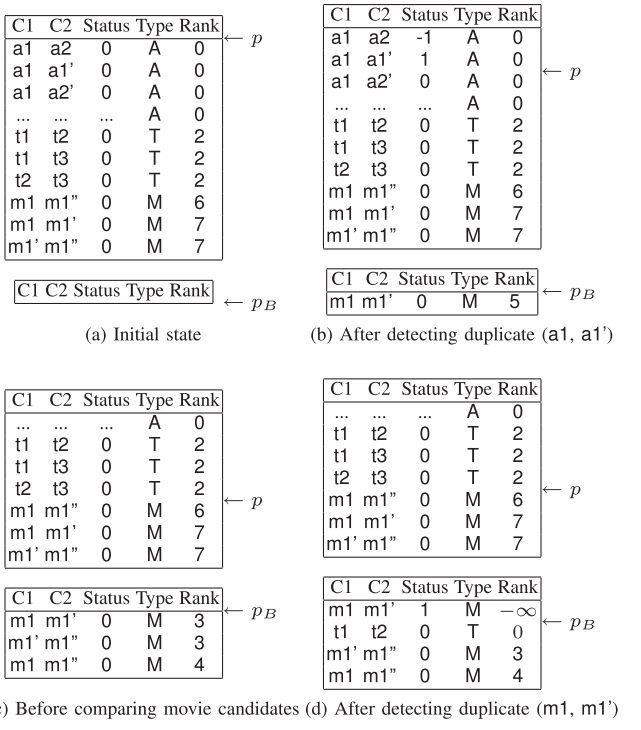


Fig. 5. Different states of pair sequence retrieved from *PQT* (top) and buffer B (bottom) using RECUS/BUFF.

Now, during retrieval of the next pair, two cases may arise: the next pair to be retrieved either comes from the sequence of pairs retrieved from *PQT* (if the lowest rank in this sequence is lower than the lowest rank in B), or from buffer B otherwise.

In the first case, the pair to be classified is the pair p points to. As a consequence, we classify the corresponding pair, make necessary updates to the pair's tuple and to B, and move p forward. In our example, this case occurs for all comparisons among actors and titles. Fig. 5c illustrates the processing status after these comparisons.

In the second case, the pair at the current position of p_B is classified, B is updated, and p_B moves to the next pair. Note that the updates caused by the classification result of a pair in B are also performed in B. The reason for this is that in general, p is not pointing to the pair being compared, so an update in the database would require moving p to the pair to be updated, updating it, and moving the pointer back to its previous position. We avoid this expensive operation by updating the status and the rank of the classified pair within B. The pair stays in B until p has moved to that pair in *PQT* or until the buffer is flushed. In the former case, we ensure that we do not classify the pair again and we only update its status in *PQT*. Fig. 5d shows the updated version of B after $(\mathrm{m1}, \mathrm{m1}')$ has been classified as a duplicate and its dependent pair $(\mathrm{t1}, \mathrm{t2})$ has been added to B. Note that the rank of $(\mathrm{m1}, \mathrm{m1}')$ is now $-\infty$ to guarantee that new pairs added to B have higher rank and that p_B points to the correct next pair.

The above examples illustrate the iterative phase when the only necessary updates are updates to the status and the rank of pairs. Upon classifying a pair as duplicate, some algorithms merge (only one candidate is left that combines

both duplicates) or enrich (both candidates remain, but their ODs and relationships are extended) duplicate pairs. Such operations require updates in relations *EDGES* and *DEP*, and to OD attributes. These updates are not local to the priority queue but are performed on the graph itself. However, these potentially affect the set of pairs that should be in PQ or the comparison order. Such changes are handled by the buffer, again to avoid sorting *PQT* at every update.

In case of a buffer overflow, a strategy that frees buffer space has to be devised. To minimize the occurrences of a buffer overflow, and hence of sorting *PQT*, we clear the entire buffer when an overflow occurs. That is, we update all pairs in *PQT* that were buffered, remove them from B, and sort *PQT* in the next retrieval phase.

As for RECUS/DUP, the theoretical worst case requires sorting *PQT* at every iteration. However, whereas RECUS/DUP reaches the worst case when every pair is classified as a duplicate, RECUS/BUFF requires the buffer to overflow at every iteration; an unlikely event when wisely choosing the size of buffer B. In setting the buffer size one should keep in mind that 1) it must fit in main memory, 2) it should be significantly smaller than *PQT* to make sorting it more efficient than sorting *PQT*, and 3) it should be large enough to store a large number of dependent pairs to avoid sorting *PQT*.

4.3 Scaling-up Classification

Classifying a candidate pair requires a similarity computation. We first present a similarity measure template. We then discuss how to efficiently compute similarity using hybrid similarity computation and early classification, methods that can be used whenever a similarity measure conforms to our template.

4.3.1 Similarity Measure Template

The similarity measure template we propose fits many similarity measures used for duplicate detection among several types of entities. For instance, [4], [6], [15], [21], [22] use one or more similarity measures that conform to the template. In the latter case, similarity measures are for instance combined by addition or multiplication. Some similarity measures are not covered by this template, e.g., [27] or rule-based classifiers [25]. But we observe that this template fits all the current measures used for iterative DDG we are aware of. Overall, we believe that our template is very useful to application developers as it guides them in designing measures that consider relationships. Also, conforming to this template enables the use of the efficient and scalable classification techniques discussed here.

Definition 7 (Duplicate influencing candidate pairs). *Given candidates (c, c'), the set of duplicate influencing candidate pairs is* $N_{ip}^{\approx}(c, c') := \{(i, i') \in I(c, c') | i, i' \text{ are duplicates}\}$.

Definition 8 (Nonduplicate influencing candidates). *Denoting an empty entry by $\perp$, the set of nonduplicate influencing candidates is*

$$N_{ip}^{\neq}(c, c') := \{(i, \perp) | i \in I(c) \wedge i \text{ has no dups in } I(c')\} \cup \{(\perp, i') | i' \in I(c') \wedge i' \text{ has no dups in } I(c)\}$$

Both definitions assume that we know whether i and i' are duplicates or not. This knowledge is acquired during the iterative phase and causes the similarity to increase.

Example 5. Assuming duplicate actor and title candidates have been detected, $N_{ip}^{\approx}(\mathrm{m1}, \mathrm{m1''}) = \{(\mathrm{a1}, \mathrm{a1''})\}$, and $N_{ip}^{\neq}(\mathrm{m1}, \mathrm{m1''}) = \{(\mathrm{a2}, \perp), (\mathrm{t1}, \perp), (\perp, \mathrm{a3'}), (\perp, \mathrm{t3})\}$.

We further introduce a weight function $w_{ip}(S)$, which captures the relevance of a set S of candidate pairs. This function has properties that allow incremental computation and that guarantee that the similarity function monotonously increases. In practice, count or variations of the inverse document frequency are used as weight function.

We make analogous definitions to compute $N_{od}^{\approx}$, $N_{od}^{\neq}$, and $w_{od}(S)$ for ODs. Opposed to duplicates in $N_{ip}^{\approx}$ that are detected using the similarity measure used for classification (i.e., sim), duplicate ODs in $N_{od}^{\approx}$ are detected with a secondary similarity distance, e.g., edit-distance. As it does not vary during the iterative phase, it can be precomputed.

Example 6. Movies have no OD attributes, so $N_{od}^{\approx}(\mathrm{m1}, \mathrm{m1''}) = N_{od}^{\neq}(\mathrm{m1}, \mathrm{m1''}) = \emptyset$. Considering title candidates t1 and t2, we have $N_{od}^{\approx}(\mathrm{t1}, \mathrm{t2}) = \emptyset$ and $N_{od}^{\neq}(\mathrm{t1}, \mathrm{t'2}) = \{(\mathrm{Troy}, \perp), (\perp, \mathrm{Troja})\}$.

Definition 9 (Similarity measure template). *The template, where all operands are optional, is defined as:*

$$sim(c, c') = \frac{w_{od}(N_{od}^{\approx}) + w_{ip}(N_{ip}^{\approx})}{w_{od}(N_{od}^{\approx}) + w_{ip}(N_{ip}^{\approx}) + w_{od}(N_{od}^{\neq}) + w_{ip}(N_{ip}^{\neq})}.$$

When all operands in the denominator are not used, sim returns the result of the nominator, thus covering similarity measures that consider only shared information ($N_{ip}^{\approx}$ or $N_{od}^{\approx}$). We omitted the parameters c and c' for brevity.

Example 7. $sim(m1, m1'') = \frac{0+1}{0+1+0+4} = 0.2$, when using count() as weight function.

A more practical similarity measure that combines two measures that comply to our template is used by RC-ER [22]. The proposed measure applies to pairs of candidate clusters instead of candidate pairs, but assuming these clusters are merged to a single candidate, our template still applies to this scenario. The similarity measure is defined as

$$sim(\zeta, \zeta') = (1 - k) \times sim_{att}(\zeta, \zeta') + k \times sim_{graph}(\zeta, \zeta'),$$

where k is a constant, ζ and ζ' are the compared sets of candidates (now merged), sim_{att} computes the similarity of ODs, and sim_{graph} computes the similarity of influencing candidate sets. For sim_{att}, Bhattacharya and Getoor [22] propose to use the SoftTFIDF score [28] as similarity measure. This similarity measure can be viewed as implementing $w_{od}(N_{od}^{\approx}(\zeta, \zeta'))$ and the denominator is not defined. In considering relationships among candidates, Bhattacharya and Getoor [22] compute the neighborhood similarity, defined as $\frac{|\zeta.N \cap \zeta'.N|}{|\zeta.N \cup \zeta'.N|}$, where $\zeta.N$ is the multiset union of the influencing candidate sets of all candidates within ζ. The neighborhood similarity complies to our template where $w_{ip} = 1$.

4.3.2 Hybrid Similarity Computation

We now describe a technique, called hybrid similarity computation, to compute a similarity that conforms to our template. Note that we focus the discussion on ODs and make some remarks concerning influencing candidate pairs.

In the hybrid version of OD weight computation, we use two SQL queries Q_1 and Q_2. Q_1 determines the set of similar OD attribute pairs with their weight. Q_2 determines the set of *all* OD attributes defined as $\mathrm{OD}(c) \cup \mathrm{OD}(c')$.

To determine the difference of ODs, we then check whether an attribute value returned by Q_2 is in the set of similar attribute values. This check is performed outside the database in an external program, as well as weight aggregation. Algorithm 1 describes the steps the external program performs.

Algorithm 1: Hybrid OD similarity computation

D: set of duplicate descriptions, initially empty;
N: set of non-duplicate descriptions, initially empty;
foreach *tuple* $\langle v_1, v_2, w\rangle$ *returned by* Q_1 **do**
  $D := D \cup \{((v_1, v_2), w)\}$;
foreach *tuple* $\langle v, w\rangle$ *returned by* Q_2 **do**
  if $\{(v_1, v_2) | ((v_1, v_2), *) \in D \wedge (v_1 = v \vee v_2 = v)\} = \emptyset$ **then**
    $N := N \cup \{((v, \perp), w)\}$;
Compute aggregate weights $w_{od}(D)$ and $w_{od}(N)$;

Example 8. When comparing the ODs of $(a1, a1')$, Q_1 returns similar value pairs, which are added to D. Hence, $D = \{((Brad\ Pitt, Brad\ Pit), 1.0)\}$. Q_2 returns $\{(Brad\ Pitt, 1.0), (Brad\ Pit, 1.0)\}$. Because both OD values are part of a similar pair in D, $N = \emptyset$.

For influencing pairs, the hybrid strategy is slightly different: based on relation *DEP* generated during initialization, we issue a SQL query Q_3 to return $I(c, c')$ in ascending order of the influencing pairs' status (duplicates come first). The external program then splits the result of Q_3 into a duplicate set D and a nonduplicate set N. The order chosen guarantees that all duplicates are added to D before the first nonduplicate appears, so that we can compute N as we do for ODs.

Example 9. When comparing $(m1, m1')$, Q_3 returns tuples $\langle a1, a1', 1.0, 1\rangle$, $\langle a2, a2', 1.0, 1\rangle$, $\langle a1, a3, 1.0, 0\rangle$, $\langle a2, a1', 1.0, 0\rangle$, $\langle a2, a3, 1.0, 0\rangle$, $\langle t1, t2, 1.0, 0\rangle$ in that order, where the tuple schema is ⟨c, c′, weight, status⟩. The external program iterates through these tuples and adds them to D as long as their status is 1. This results in $D = \{((a1, a1'), 1), ((a2, a2'), 1)\}$. The remaining tuples are split up in two candidates and, after checking which candidates are not already in D, we obtain $N = \{((a3, \perp), 1), ((t1, \perp), 1), ((t2, \perp), 1)\}$.

The results of queries Q_1, Q_2, and Q_3 need to be processed in main memory. But compared to the in-memory buffer, we consider this main-memory consumption as negligible, because in the worst case $|D| + |N| = |I(c)| + |I(c')|$. In practice, these sets are small (14 candidates being the maximum observed in experiments reported in DDG algorithms summarized in Section 2). The main advantage of hybrid similarity computation over similarity computation using a single monolithic SQL query is that it overcomes database system limitations such as the lack of user-defined SQL aggregate functions support and allows us to interfere with the computation process, for instance by applying early classification.

4.3.3 Early Classification

Essentially, early classification interrupts similarity computation as soon as we know if the outcome results in a duplicate or nonduplicate classification. So it helps to increase classification efficiency when a significant portion of pairs are nonduplicates. Early classification distinguishes itself from existing filters (defined as upper bounds to the similarity measure [15], [29]) in that no extra filter function is defined to prune nonduplicates prior to similarity computation. Instead, the similarity function is computed incrementally and intermediate results are used to classify nonduplicates.

If a candidate pair's similarity $sim(c, c') > \theta$, candidates c and c' are duplicates, and nonduplicate otherwise. Weight functions commonly return results greater or equal to zero and, when used for relationships, produce values greater or equal to 1 on nonempty input sets. Based on this assumption, the following inequations, which we use in our implementation, correctly classify nonduplicates, even before sim (Definition 9) is calculated completely. Should there not be a guarantee that the denominator is greater or equal to one, only the second pruning function applies

$$w_{od}(N^{\approx}_{od}) + w_{ip}(N^{\approx}_{ip}) \leq \theta \rightarrow sim \leq \theta$$
$$\frac{w_{od}(N^{\approx}_{od}) + w_{ip}(N^{\approx}_{ip})}{w_{ip}(N^{\neq}_{ip}) + w_{od}(N^{\approx}_{od}) + w_{ip}(N^{\approx}_{ip})} \leq \theta \rightarrow sim \leq \theta.$$

The larger the number of influencing pairs or OD attribute values of candidates are, the more processing is potentially saved using early classification, because the number of iterations through nonduplicates among influencing pairs and ODs (i.e., in the results of Q_3 and Q_2, when it was executed) that are potentially saved increases.

5 PDDG: Parallel and Batched DDG

The methods presented so far iterate over pairs of candidates, one pair at a time. In this section, we introduce a framework that allows processing multiple pairs at the same time, using parallelization and batched processing. This extended framework, called *parallel* DDG, or PDDG for short, enables the comparison of sets of candidate pairs (batches) and the distribution of multiple DDG processes over multiple processing units (parallelization).

We first introduce some additional definitions. As we show, the update phase is the only one that significantly changes from DDG to PDDG, and we describe these modifications after introducing the framework.

5.1 PDDG Framework

We employ a client-server approach; we name the server *job manager* and clients are called *workers*. The job manager assembles work-packages and distributes them to the workers.

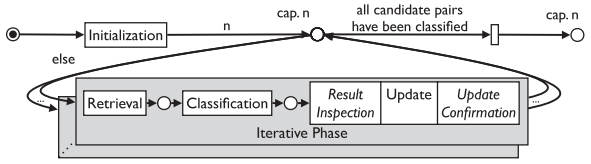


Fig. 6. General workflow of PDDG.

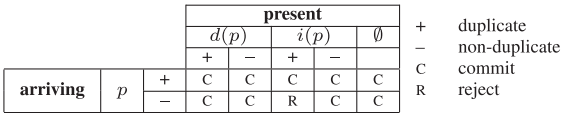


Fig. 7. Sample decision matrix for update instructions.

Definition 10 (Work-package). *A work-package w of size s is a set of candidate pairs: $w = \{(c_{11}, c_{12}), \ldots, (c_{s1}, c_{s2})\}$.*

How the job manager distributes work-packages is algorithm-specific. For graphs with high connectivity, which potentially results in many recomparisons, it is advisable to cluster pairs that depend on one another into one work-package to reduce communication overhead or to avoid distributing these in parallel. In our implementation, we do not consider such optimizations as the benefit of parallelization significantly outweighs the benefit of saving some recomparisons. Instead, we initially sort pairs based on their rank, iterate over these and create a new work-package every $|w|$ pairs.

Workers process the work-packages and report their classification results to the job manager, which processes the results and distributes new work-packages to idle workers. The client-server approach limits the complexity of our system and reduces communication effort. The only instance that performs synchronization is the job manager. Workers perform independently from each other, even though the computations are potentially interdependent due to the fact that work-packages, similar to candidate pairs, can influence or depend on other work-packages.

Definition 11 (Dependent & influencing work-packages). *The set of dependent work-packages of a work-package w is the set of work-packages that contain at least one dependent pair $p_D \in D(p)$ of a pair $p \in w$. We define the set of influencing work-packages analogously.*

Fig. 6 illustrates the PDDG framework. Similarly to DDG, an initialization phase prepares the database and performs some precomputations. Afterwards, n workers are instantiated and the parallel execution starts. Each worker performs the actions defined in the iterative phase for the work-package assigned to it. Once all workers finish their jobs and all candidate pairs have been classified, the PDDG algorithm terminates. The steps of the iterative phase of PDDG are very similar to the steps of the nonparallel DDG, except for the update phase, which requires two additional steps, namely *result inspection* and *update confirmation*. Therefore, we focus the subsequent discussion on these two steps. Another subtle difference is that the different steps handle work-packages, not pairs.

5.2 Handling Updates in PDDG

Before we discuss the details of the additional update steps, let us first describe the problem that makes these necessary.

5.2.1 Update Synchronization

In DDG, candidate pairs with classification status *nonduplicate* (-1) are reset to *unknown* (0) if an influencing candidate pair has been classified as duplicate. Pairs marked with classification status *unknown* are placed (back) into the priority queue, to ensure that the pair is compared again as its similarity may have increased.

When introducing parallelization, we have to consider the following: Let p_1 and p_2 be two pairs that have to be classified and $p_1 \in D(p_2)$ holds. Let both pairs be computed in parallel, e.g., p_1 by processor P_1 and p_2 by P_2. Now, assume p_2's computation ends first. Having been classified as duplicate, the *dependent* pair p_1 is reset to *unknown*. After that (or even in parallel) p_1's computation completes and its state is set to *nonduplicate*, thus overriding the reset information. In this case, the execution order $P_1|P_2$ yields a final state of nonduplicate for p_1, whereas the order $P_2|P_1$ would yield an unknown state for p_1, requiring the recomparison of p_1. Note that the latter result is the correct one.

This example shows that we have to take special care in implementing the update phase in PDDG so that we can guarantee the equivalence of parallel execution. In general, a parallel computation of two pairs p_1 and p_2 with $p_1 \in D(p_2)$ can be performed *correctly* if 1) their interdependency is *recognized* and 2) their read and write operations can be *synchronized* in a manner that they occur as if p_1 and p_2 are processed sequentially with respect to their interdependencies.

5.2.2 Implementing Update in PDDG

The job manager is the only entity in our framework that knows about all other entities (the workers); both recognizing interdependencies and deciding how to synchronize updates is thus implemented by the job manager.

To recognize interdependencies, the job manager maintains a *registry* that contains information about all candidate pairs that need to be updated and their classification results. The job manager is further capable of recognizing interdependencies between candidate pairs in the registry. This information allows decisions on how to proceed with a classification result reported by a worker.

More precisely, a worker first sends the classification results of all pairs in its work-package to the job manager. Using the registry, the job manager then checks each classification result for potential conflicts. Based on the result of this analysis, the job manager finally decides how to proceed with a classification result. Possible decisions are: 1) *Commit* (the result shall be committed to the database), 2) *Reject* (the result is obsolete and should be discarded and not written to the database.), and 3) *Pending* (the result should be committed but the commitment needs to be delayed). The reason for delaying a commit may be conflicting write operations on a shared memory, e.g., a nonsynchronized file access or a nonsynchronized queue.

Each time a candidate pair is *committed*, it is saved in the registry until a worker sends an update acknowledgment

during the update confirmation phase. This acknowledgment forces the job manager to remove it from the registry.

To decide whether to commit, reject, or delay an update, the job manager relies on an update decision matrix. In general, the update matrix is specific for a given underlying algorithm. Fig. 7 shows a decision matrix that applies to algorithms that do not simultaneously distribute the same pair multiple times and that trigger the recomparison of pairs by setting their status to unknown. This decision matrix may for instance be used to parallelize RECUS/BUFF or [4], [5]. For this type of algorithm, the only type of conflict that potentially occurs is the one described in Section 5.2.1. That is, the job manager has to ensure that when a pair p has been classified as a nonduplicate after an influencing pair has been classified as duplicate in parallel, the result for p is obsolete and should be rejected so that the status of p remains unknown and p gets compared again.

The matrix of Fig. 7 is organized as follows: A pair p has been classified and an update instruction based on the information of the registry has to be devised. The classification result of p is depicted as $+$ (duplicate) or $-$ (nonduplicate) in the two rows. Possible relationships between pairs in the registry and p are shown in the column heads: $d(p)$ means there is at least one pair in the registry that depends on p; $i(p)$ means there is at least one pair in the registry that influences p; and $\emptyset$ means none of the above applies. Depending on the order of arriving pairs at the registry, different decisions are made. If more than one combination is possible, e.g., p has dependent as well as influencing pairs in the registry, the most conservative decision is chosen (the ascending order of conservative decisions is *commit*, *pending*, *reject*).

We show that the above decision matrix is correct for the type of algorithm it is meant for, i.e., algorithms where a pair is considered by at most one worker in parallel, and where recomparisons are triggered by (re)setting the status of pairs to unknown. First, it is easy to see that when p has no dependent or influencing pairs in the registry, the corresponding classification result can simply be committed. Therefore, column 5 is correct. Let us now consider the case where the registry already contains pairs that depend on the classification result of p. For those dependent pairs, the status of nonduplicate dependent pairs has to be reset to unknown but the result for p can simply be committed as correctly described by the decision matrix (columns 1 and 2). In our implementation, the job manager adds pairs that have to be recomputed to an in-memory queue extension without affecting their persistent status, thus implicitly setting their status to unknown. Let us now consider the cases where the registry contains at least one influencing pair of p, i.e., columns 3 and 4. If p is classified as duplicate, this result can simply be committed as the albeit outdated information used for classification was sufficient to determine the duplicate p (row 1 in both $i(p)$ columns). In case the influencing pairs in the registry are nonduplicates, the classification of p is not based on outdated information and hence, the result can also be committed (row 2, column 4). Finally, if an influencing candidate pair present in the registry is a duplicate, we have to assume that the classification of p did not consider this information, which may yield a wrong classification as nonduplicate. In this case, we reject the nonduplicate status of p, implying that its status remains unknown.

The decision matrix does not illustrate the pending state. One example where this state is reasonable occurs when a pair p may be distributed to more than one worker simultaneously. This would require two additional columns in the decision matrix. As long as the more recent classification result is the same as the previous one (i.e., p is already in the registry), the new result can be rejected as nothing changes compared to the previous result. Similarly, if the older result classifies p as a duplicate but the more recent one does not, we keep the duplicate and reject the nonduplicate status of the arriving pair. Finally, assume the registry contains p as a nonduplicate, a result to be committed, and a new incoming pair designates p as a duplicate. In this case, the job manager has to ensure that the nonduplicate status is processed before the duplicate status, otherwise, the duplicate status may be overwritten. To this end, a pending status is assigned to the duplicate.

During the update phase, the worker follows the job manager's instructions and either persists a classification result (*commit*) or just discards it (*reject*). All candidate pairs with classification status *nonduplicate* that are dependent on successfully persisted duplicate pairs are requested for recomputation, as their similarity has potentially increased.

The final phase is update confirmation. After all updates have been processed, the worker sends a confirmation message concerning the persisted and discarded pairs to the job manager. Finally, some cleanup tasks are performed and the next iteration is prepared if there is work left to process. Otherwise, the worker finishes computation and suspends.

5.3 Batched DDG Using the PDDG Framework

Apart from the parallelization of DDG, PDDG introduces batch processing: the smallest unit of processing is no longer a candidate pair, but a work-package. Batch processing can reduce runtime for both sequential and parallel DDG execution.

In the sequential execution of DDG, the runtime reduction comes from batch updates to the database. Indeed, instead of sending an update request for each individual pair, we send only one update request for a batch of pairs, i.e., all pairs in a work-package. It is commonly known that batch updates can be performed more efficiently by a DBMS than a sequence of several, albeit smaller update transactions.

In the presence of parallelization, apart from the fact that batch updates can be used, enlarging the size of work-packages leads to less waiting conditions. If there was only one pair per work-package, workers had to request the job manager for each pair. Therefore, it is not uncommon that multiple workers request new work at the same time. Clearly, the job manager then becomes a bottleneck and requests are enqueued.

Using batch processing, multiple pairs within a work-package can make it dependent on itself, i.e., if both $p_1, p_2 \in w$ and $p_1 \in D(p_2)$. In this case, if p_2 is classified before p_1, the computation of p_1 does not recognize the classification result of p_2. This is because the classification result of p_2 has not been made persistent: it is first saved in a worker-internal buffer and later updated to the database. We studied different strategies that may be used to address this problem, however,

we do not elaborate on these due to space constraints. In general, the best strategy depends on the data that are being processed; runtime is influenced by the duplicate ratio, the data connectivity, and the similarity measure. An interesting aspect is the influence of comparison order because recomparisons significantly reduce efficiency when the reference graph has high connectivity [5]. Batch processing may mix up the order in which candidate pairs are compared, because rerequests of pairs are done after *all* pairs within the workpackage have been classified. Thus, in presence of reference graphs with high connectivity, batch processing is useful only to reduce runtime if time savings due to fewer updates outweigh the time that is needed to perform additional recomparisons. In our implementation, we opted for an optimistic strategy that simply ignores interdependencies and runs the risk of causing a recomparison of dependent pairs. This strategy is suited for graphs with moderate connectivity and duplicate ratios.

6 Evaluation

We first discuss how DDG (without parallel and batched processing) scales depending on various parameters, using the discussed algorithms. We then present experiments that show the effect of parallel and batched processing on runtime. We use both artificial and real-world data.

6.1 Data Sets

Artificial movie data (ArtMov). From a list of 35,000 movie names and 800,000 actors from `imdb.com`, we generate data sets for which we control

1. the number of candidate movies M and the number of candidate actors A to be generated,
2. the connectivity c, i.e., the average number of actors that influence a movie and vice versa,
3. the duplicate ratio $dr \in [0, 1]$, defined as the percentage of duplicate pairs that enter the priority queue,
4. the probability of errors in ODs consisting of a movie name and an actor name for the respective candidate types, and
5. the probability of errors in influencing candidates. Errors in ODs include typographical errors (adding or removing single characters) and contradictory values (removing half the characters in a string, which yields our string similarity measures incapable of recognizing them to be similar).

Errors in influencing candidates include the removal of edges. Further details are available in [26].

We generate the same number of actor and movie candidates: $A = M$. With these, we generate PQ of size $|PQ|$ that contains the same number of actor and movie pairs. PQ has duplicate ratio dr, hence, we generate $dr \cdot |PQ|$ duplicates and the remaining $|PQ|(1 - dr)$ pairs in PQ are nonduplicates. The generation of these two sets is based on an original set of k duplicate-free candidate actors and k duplicate-free candidate movies. To generate duplicate pairs, we simply copy originals and introduce errors in the copy. We copy an original at most once. To generate a sufficient number of duplicates, we need at least $k_D = \frac{|PQ| \cdot dr}{2}$ original candidates of one type. Nonduplicate pairs are generated based on k_O original actor and k_O original movie candidates, so we require k_O s.t. $\frac{|PQ|(1-dr)}{2} = \frac{k_O \times (k_O - 1)}{2}$. The number of original duplicate-free movie (and actor) candidates is then given by $k = \max(\lceil k_D \rceil, \lceil k_O \rceil)$. In total, the number of candidates of each type equals the number of originals k plus the number of duplicates of a given type, which is equal to $k_C = \frac{|PQ| \times dr}{2}$. Therefore, the number of candidates and the size of PQ, which we both show in our experiments, correlate based on

$$A = M = k + k_C = \max\left\{ \left\lceil \frac{|PQ| \cdot dr}{2} \right\rceil, \lceil k_O \rceil \right\} + \left\lceil \frac{|PQ| \cdot dr}{2} \right\rceil.$$

Error probabilities are set to 20 percent. When not mentioned otherwise, connectivity $c = 5$ and buffer size is 1,000.

Real-world CD data (RealCD). The real-world data set is CD data from `freedb.org`. We consider CDs, artists, and track titles as candidates. CD ODs consist of title, year, genre, and category attributes, and they depend on artist and track candidates. Artist and track candidates, respectively, have the artist's name and the track title as OD. Track candidates depend on artist candidates. A data set consisting of approximately 10,000 CDs with manually labeled duplicates is available at our repeatability website (http://www.hpi.uni-potsdam.de/naumann/projekte/repeatability/). In our experiments, we process one million candidates.

Evaluation focus. We deliberately focus on experiments evaluating the scalability of the iterative phase of DDG. That is, runtime of the initialization phase is not reported in our experiments. Benefits of DDG on effectiveness have already been studied extensively (see Section 2). For ArtMov data generated the same way as described in this paper, we observed an improvement on the F-measure when using DDG of up to 29 percent [24] (depending on the connectivity). Using DDG, the F-measure on our hand-labeled sample of 10,000 CDs improves by 41 percent compared to an algorithm that considers ODs only (i.e., title, year, genre, and category) using the same OD similarity measure (unit weights and similar values having a normalized string edit distance smaller than 0.3). Arguably, this may not be the optimal configuration for the variant not using relationships, but we opted for the same OD configuration as for DDG to enable direct comparison.

6.2 Experimental Evaluation of DDG

We begin with an evaluation of the algorithms proposed for DDG in Section 4. For this series of experiments, we used DB2 V8.2 as DBMS, running on a Linux server and remotely accessed by a Java program running on a Pentium 4 PC (3.2 GHz) with 2 GB of RAM. That is, all runtimes reported also include network latency. To obtain a competitive database configuration, we employed the DB2 configuration advisor using our specific workload. We repeated experiments five times to obtain average execution times.

6.2.1 Retrieval and Update Scalability

We first compare the scalability of RECUS/BUFF to the scalability of the baseline algorithm RECUS/DUP.

Experiment 1. We start with an evaluation of how both algorithms behave with varying dr on various data set sizes.

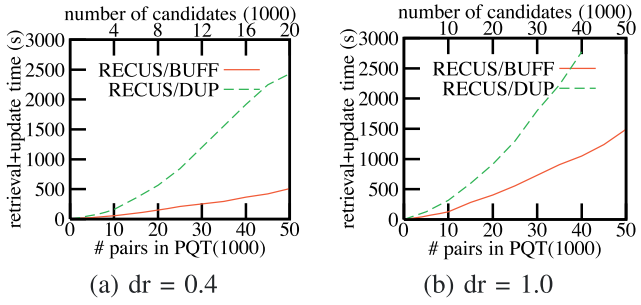


Fig. 8. Retrieval and update time, varying $|PQ|$ and dr.

Methodology. We generate ArtMov data with table PQT (that stores PQ) ranging from 10,000 to 50,000 candidate pairs in increments of 5,000, and vary the duplicate ratio dr between 0.2 and 1.0 in increments of 0.2. As representative results, we show runtimes of RECUS/DUP and RECUS/BUFF (in seconds) for dr values of 0.4 and 1.0 in Fig. 8. The number of candidates ($A + M$) is shown at the top x-axis, the bottom x-axis shows the size of PQT (in thousands).

Discussion. Fig. 8 clearly shows that RECUS/BUFF outperforms RECUS/DUP regardless of dr: Obviously, sorting PQT is more time consuming than maintaining the order of the smaller in-memory buffer B. We further observe that the higher dr, the more time retrieval and update needed for both algorithms, because in both algorithms sorting PQT occurs more frequently: RECUS/DUP sorts PQT every time a duplicate has been found, and RECUS/BUFF sorts the PQT every time B overflows, happening more frequently, because influencing neighbor pairs enter B more frequently. The final observation is that with increasing priority queue size/number of candidates, RECUS/BUFF scales almost linearly for practical duplicate ratios (below 0.8). Therefore, we expect RECUS/BUFF to be efficient even on very large data sets as Experiment 6 confirms.

Experiment 2. From Experiment 1, we conclude that RECUS/BUFF performs better than RECUS/DUP, because it sorts PQT less frequently, and instead maintains a main-memory buffer B of fixed size b. Clearly, b plays a central role in the efficiency gain. Another factor that affects the filling of B is the connectivity c. The higher it is, the more neighbors enter B when a duplicate is found, and an overflow occurs more frequently. So, we expect RECUS/BUFF to be slower with increasing c and smaller b.

Methodology. We study how a changing buffer size affects ArtMov data with 10,000 pairs in PQT and $dr = 0.4$. We vary b from 1 to 10,000. We further vary connectivity c from 1 to 5 for all considered buffer sizes. Results are shown in Fig. 9, which depicts the sum of retrieval and update time for all buffer sizes (left), update time for small buffer sizes (top), and retrieval time for small buffer sizes (bottom).

Discussion. We observe that for all but very small b, both retrieval and update time stabilize. Furthermore, for studied connectivities, the runtime increases linearly with c. The increase in runtime is mainly due to the increased update complexity: for larger c, when a duplicate is found, a larger number of dependent pairs needs to be determined and added to B. As a general rule of thumb, a buffer size of 1,000 suffices to significantly improve efficiency.

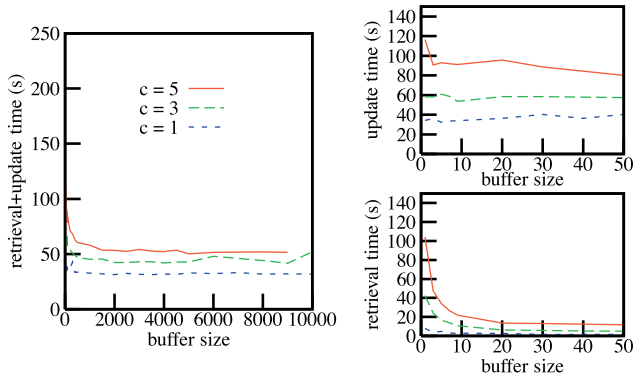


Fig. 9. Retrieval and update time for varying b and c.

6.2.2 Classification Scalability

We evaluate the scalability of classification, considering a baseline SQL-based approach, we call SQL/Complete (SQL/C for short) that computes the similarity using a monolithic SQL query [26] and hybrid similarity computation with and without early classification, called HYB/Complete (HYB/C, see Section 4.3.2) and HYB/Optimized (HYB/O, see Section 4.3.3), respectively. We use RECUS/BUFF as DDG algorithm.

Experiment 3. We compare runtimes of SQL/C, HYB/C, and HYB/O on ArtMov data of varying size and dr.

Methodology. We generate ArtMov data with PQT sizes ranging from 5,000 to 40,000 in increments of 5,000, and for each size, we generate data with dr varying from 0.2 to 1.0 in increments of 0.2. We run each classification method on each data set, and measure its runtime. Results are shown in Fig. 10 for duplicate ratios $dr = 0.2$, and $dr = 0.8$.

Discussion. SQL/C and HYB/C have comparable execution times, because they both have to compute the same result, albeit using different strategies. All classification methods scale linearly on the range of considered priority queue sizes, and hence with the number of candidates when no blocking technique is additionally used. Early classification allows to save classification time: at $dr = 0.2$, 32 percent of classification time (compared to HYB/C) is saved, which gracefully degrades to 26 percent at $dr = 0.8$, when 40,000 pairs are compared. For $dr = 1$, we still observe 5 percent savings, which are due to pairs that are classified as nonduplicates, although they are in fact duplicates. The reduction in the benefit of early classification with increasing dr is due to the fact that the more duplicates exist in PQT, the fewer similarity computations may be aborted for nonduplicates.

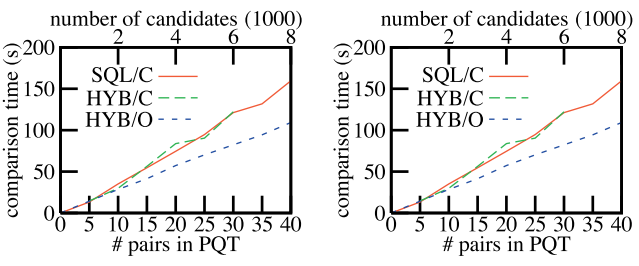


Fig. 10. Classification time comparison.

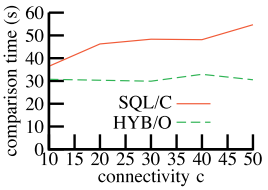


Fig. 11. Classification time & connectivity.

Experiment 4. We now study how connectivity c affects classification efficiency. Because c defines how many influencing neighbors a candidate has, which have to be determined using join operations and processed, we expect similarity computation to be slower for larger c when using SQL/C or HYB/C. On the other hand, HYB/O potentially saves more processing the larger c.

Methodology. We vary c from 10 to 50 in increments of 10 for an ArtMov priority queue of size 10,000 and $dr = 0.4$. Fig. 11 reports comparison times for SQL/C and HYB/O, SQL/C and HYB/C being comparable.

Discussion. When using SQL/C, comparison time increases with increasing c, an effect also observed by other DDG algorithms. On the other hand, runtime is around 7 ms per candidate for all c when using HYB/O. This experiment shows that HYB/O counters the negative effect of increasing c on efficiency. We currently do not have an explanation for the shape of the SQL/C curve, where comparison time is roughly constant between $c = 20$ and $c = 40$. We suspect that "intriguing behavior of modern [query] optimizers" [30] is partly responsible for that behavior. Nevertheless, the general trend is clear.

The analysis of the techniques proposed in this paper using artificial data of moderate size leads to the conclusion that RECUS/BUFF and HYB/O scale-up best.

6.2.3 Real-World Behavior

Experiment 5. We now study real-world scalability of RECUS/BUFF in combination with HYB/O on a large data set.

Methodology. Using RealCD data with 1,000,000 candidates, we apply a blocking technique to reduce the number of candidate pairs entering *PQT*, a common technique also used by Bhattacharya and Getoor [6], Kalashnikov and Mehrotra [10]. More specifically, we use the Sorted Neighborhood Method (SNM) to consider only candidate pairs [16] within a window of size 3 (value chosen based on experience). As sorting keys, we use the first four consonants of an artist's name for artist candidates, the first six characters of a track title for track candidates, and the first four consonants of an artist's name plus the last two digits of a year as key for CD candidates. Note that SNM affects only the set of pairs entering the priority queue, but the data graph remains unchanged. That is, a similarity comparison still considers *all* influencing candidates during classification. During update, we do not add new candidate pairs to *PQT*. After blocking, 2,000,000 candidate pairs enter *PQT*. Buffer size is 1,000, so that we can compare runtimes with those obtained on artificial data. We observe that retrieval takes 1,379 s, classification takes 5,482 s, and update takes 17,572 s.

Discussion. Among the two million candidate pairs in *PQT*, we found 600,000 duplicates among all candidate types, so the observed duplicate ratio is 0.3. If we extrapolate retrieval and update time obtained on ArtMov data of size 50,000 with $dr = 0.3$, for which we obtained a retrieval time of 36 and 432 s for update using a buffer size of 1,000, we see that the results obtained on a million candidates with similar parameters are in accord with the linear behavior of retrieval and update observed in Experiment 1. Indeed, the expected retrieval time is 1,440 s and we observed 1,379 s. Similarly, the expected 17,280 s for update and the 17,572 s measured are only 2 percent apart. Classification time is also in accord with the linear behavior of HYB/O observed in Experiment 3.

We see that update requires the most time. As we see in our evaluation of PDDG this time can significantly be reduced when using batched DDG, even without parallelization.

6.3 Experimental Evaluation of PDDG

We now experimental evaluate the effect of parallelism and batched processing introduced by PDDG. We run PDDG locally on an IBM X3500 system including two quad-core CPUs (Intel Xeon 2.4 GHz), 16 GB RAM, and RAID/5 with six hard drives (15,000 rpm) for parallel I/O. The server is running an Ubuntu Linux 8.04 64 Bit server edition (SMP kernel) and DB2 v9.5 64 Bit is used as DBMS. The database is locally accessed by a Java 1.5 program. For parallel computation Java Threads are used. Again, we generate ArtMov data for our experiments. By default, we use a duplicate ratio of 0.2 and an average data connectivity of 2.

When using PDDG, values for 1) the number n of parallel processing workers and 2) the size s of a work-package must be defined (within an experiment we assume all work-packages to be of same size). We denote a PDDG configuration as PDDG(n, s). A well-balanced configuration for n and s is hard to guess, because the parametrization depends on the system it runs on. The following experiment considers the parametrization of PDDG and the effect of the size of the priority queue PQ.

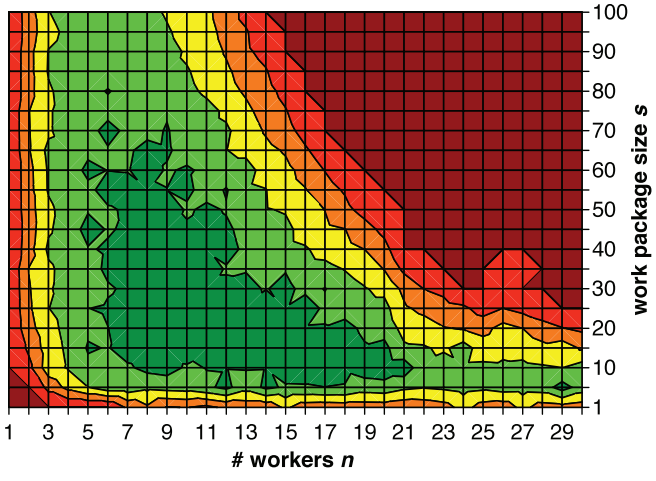
Experiment 6. To determine a well-balanced configuration for n and s, runtimes of several configurations of PDDG are compared. In addition, we repeat all experiments with different priority queue sizes. The goal of this experiment is to study the relationship of n and s for varying sizes of PQ and to establish a good configuration for further experiments.

Methodology. For different $|PQ|$ values ranging from 8,000 to 128,0000 candidate pairs, we generate artificial data. For each test case we vary n (from 1 to 30) and s (from 1 to 100). Fig. 12 depicts the resulting runtimes encoded as colored areas for the different PDDG(n, s) configurations. We create five intervals for runtimes, which linearly scale from the lower bound, which is given by the minimum runtime, to the upper bound, which is given by the average runtime. These intervals are mapped to different colors, which form five zones and fade from green (close to the minimum runtime) to red (average of all measured runtimes). Results above the average runtime are not separated and are illustrated as dark red.

Discussion. The diagrams in Fig. 12 have in common that there is a characteristic dark green triangular shape. This is the area where the best runtimes were achieved.

There are two reasons for the increasing runtimes above the hypotenuse: There are 1) larger waiting conditions due to the increasing number of workers that have to be

(a) PQ size: 8,000
min./avg. runtime: 1.7/5.0 s



(b) PQ size: 128,000
min./avg. runtime: 21.8/65.0 s

Fig. 12. Runtime of varying PDDG configurations.

enqueued by the job manager and 2) there is an increasing effort for logging and synchronizing. As a result, we approximate the best fitting configuration of PDDG on our test system as $\mathrm{PDDG}(10, 30)$, which is used as the reference scenario in the next experiments.

Increasing the size of PQ does not have an effect on the relationship between n and s besides the fact that the dark green area slightly shrinks. The larger PQ is, the more often unfavorable conditions for workers[1] appear during processing, which in turn cause more moderate runtimes and lower efficient values (values that fall in the dark green zone). This is because the variance of runtimes with small priority queue sizes is much smaller than those of larger priority queue sizes.

Experiment 7. In this experiment we investigate 1) the overall effect of applying batch processing, 2) the overall effect of applying parallelization, and 3) the effect of a combination of both in comparison to the best DDG algorithm. In addition, we evaluate the differences in runtimes of the DDG algorithm and the sequentially parameterized PDDG algorithm, $\mathrm{PDDG}(1, 1)$, when varying the priority queue size.

1. Inappropriate load balancing causes more waiting conditions (many workers request work at the same time and are enqueued) or increasing response times from the DBMS (many workers access the database at the same time).

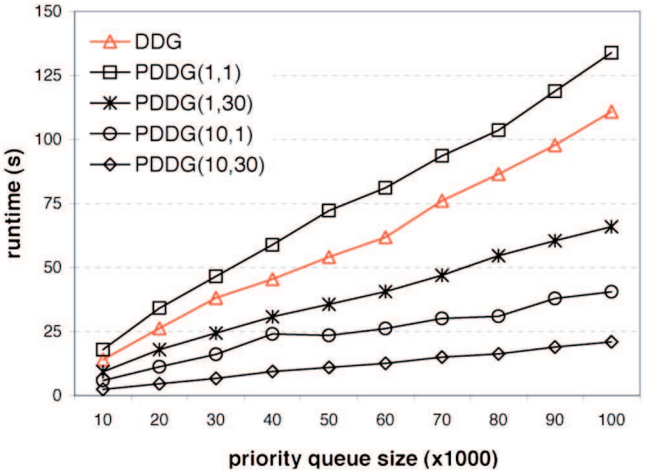


Fig. 13. Comparison of DDG and PDDG runtimes.

Methodology. For sizes of the priority queue ranging from 10,000 to 100,000 pairs we measure the runtime of the DDG algorithm, the sequential PDDG algorithm ($\mathrm{PDDG}(1, 1)$), the sequential PDDG algorithm with extensive batch processing ($\mathrm{PDDG}(1, 30)$), the parallel algorithm without any batch processing ($\mathrm{PDDG}(10, 1)$), and a combination of a parallelized PDDG algorithm with batch processing ($\mathrm{PDDG}(10, 30)$). Fig. 13 depicts the results of this experiment.

Discussion. First we observe that $\mathrm{PDDG}(1, 1)$ has longer runtimes than DDG. Even though $\mathrm{PDDG}(1, 1)$ is sequential, it comes with all redundancy and synchronization mechanisms such as logging and locking. Note that we observe an overall linear runtime, even in this new setting. Increasing the work-package size to 30 leads to the $\mathrm{PDDG}(1, 30)$ configuration, which already clearly outperforms DDG. With a priority queue size of 100,000 pairs DDG achieves a runtime of 111 s whereas $\mathrm{PDDG}(1, 30)$ ran only 66 s, saving 41 percent. When setting the work-package size back to 1 and increasing the number of workers to 10 ($\mathrm{PDDG}(10, 1)$), we observe a runtime saving of 64 percent for PDDG in comparison to DDG, and of 39 percent in comparison to $\mathrm{PDDG}(1, 30)$. Combining both techniques, the runtime saving of $\mathrm{PDDG}(10, 30)$ is 81 percent in comparison to DDG. Summarizing the results, both improvements—batch processing and parallelization—have an important effect on the reduction of runtimes and a well-balanced combination of both techniques ($\mathrm{PDDG}(n, s)$) is even faster than the sum of the runtime improvements of both noncombined configurations ($\mathrm{PDDG}(1, s)$ and $\mathrm{PDDG}(n, 1)$ where $n, s > 1$).

Experiment 8. Let us now evaluate whether PDDG is also applicable for large data sets as they are found in real-world applications and if the runtimes scale adequately with an increasing priority queue size.

Methodology. We run $\mathrm{PDDG}(10, 30)$ on generated test cases with PQ sizes ranging from 2 to 10 million pairs

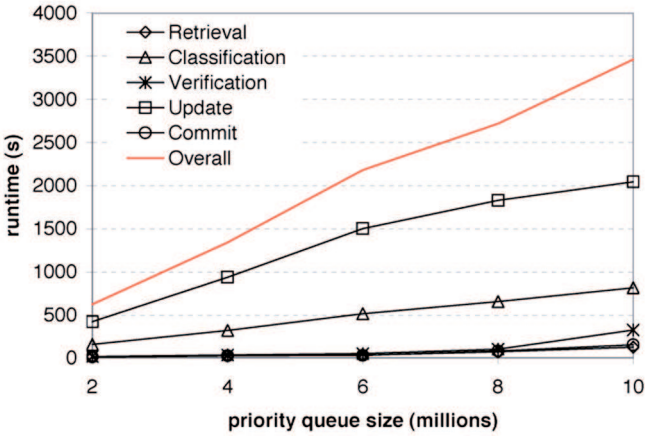


Fig. 14. PDDG runtimes on large data sets.

(which corresponds 400,000 to 2 million candidates when using our data generator) and measure overall runtime as well as the phase-wise runtimes. Fig. 14 shows the results.

Discussion. We observe that the overall runtime of PDDG does scale roughly linearly, which is in accordance with observations on smaller data sets. Using PDDG$(10, 30)$ we computed 10 million pairs in less than an hour (3,463 s). We further see that 86 percent of the overall runtime is caused by the update and the classification phases, which is due to time consuming database operations.

6.4 Comparative Evaluation

Fig. 15a summarizes how the different phases of DDG using RECUS/BUFF scale in practice depending on the data set size pq, the duplicate ratio dr, and the connectivity c. In total, we observe a roughly linear behavior for realistic parameter values throughout our experiments, even though the theoretical complexity may be higher. In studying other approaches for DDG, we observe that both RC-ER[22] and RelDC [10] do *not* scale linearly in pq and c and that LinkClus [12] does not scale linearly in pq. Dedupalog [13] reports a linear behavior in pq. Dong et al. [4] do not report any results on efficiency or scalability. Over a small data set of 1,800 candidates, R-Swoosh [14] exhibits a roughly linear behavior w.r.t. the number of processed candidates.

Fig. 15b summarizes data set sizes in terms of the number of candidates (often the only number reported) and runtime results *reported* for iterative DDG algorithms, which altogether use considerably smaller data sets than those reported here. As the experimental settings for different algorithms vary, the runtime results give only an indication of how the runtimes of different algorithms compare. Also, the different approaches use different blocking techniques to reduce the number of pairwise comparisons while maintaining effectiveness. The comparison here focuses on showing how many candidates can be deduplicated and how long this process takes. Note that runtime ranges in Fig. 15b are from different algorithm configurations.

We observe that RECUS/BUFF takes comparably long, but this comes as no surprise as database communication overhead and network latency add to the runtime. However, as we have seen in Experiment 7, using batch processing without parallelization to reduce the runtime of the update phase through batched database updates reduces the runtime by 41 percent. When parallelizing the process, we observe a runtime reduction by 81 percent compared to the runtime of DDG.

Parameter	Retrieval and Update	Classification
PQT size pq	linear (Exp. 1, 5)	linear (Exp. 3, 5)
duplicate ratio dr	linear (Exp. 1)	constant (Exp. 3)
connectivity c	linear (Exp. 2)	constant (Exp. 4)
Overall (as observed)	**linear in practice**	**linear in practice**

(a) DDG scalability using RECUS/BUFF and HYB/O in practice

Approach	# Candidates	Reported runtime (s)
Parallel R-Swoosh [14]	1,800	40 – 45
Dong05 [4]	40,516	not reported
RC-ER [6], [22]	88,070	543 – 690
DDG (RECUS/BUFF)	**1,000,000**	24,433
PDDG (10, 30)	**2,000,000**	3,463

(b) scalability of iterative DDG algorithms based on reported results

Fig. 15. Comparative evaluation.

7 Conclusion

This paper is the first to consider scalability of duplicate detection in graph data (DDG). We presented a generalization of iterative DDG algorithms consisting of an initialization phase and an iterative phase. The latter in turn consists of retrieval, classification, and update steps. We then presented how to scale up these phases with the help of an RDBMS.

For iterative retrieval and update we proposed RECUS/BUFF to scale in space and in time. It uses an internal buffer to avoid the expensive sorting performed by the straightforward baseline algorithm RECUS/DUP. To scale-up classification of candidate pairs, we proposed hybrid similarity computation to scale in space and to overcome the limitations of a pure SQL variant. To scale-up classification in time, we presented the early classification technique, which interrupts similarity computation when it is certain that a pair is not a duplicate. Experiments on large amounts of data validate our DDG approaches and show that these significantly outperform a straightforward mapping of DDG from main memory to a database.

Part of the research on scalable DDG was successfully transferred to an industry project [25]. In addition, we extended DDG to include parallel and batched duplicate detection, and presented the corresponding PDDG framework. The most significant changes to the original DDG framework occurred in the update phase. Experiments using PDDG show that using both parallel and batched processing significantly reduces the runtime compared to DDG.

Acknowledgments

This research was supported in part by the German Research Society (DFG grant no. NA 432). This work was partially done at the Hasso Plattner Institute and at IBM Almaden.

References

[1] A.K. Elmagarmid, P.G. Ipeirotis, and V.S. Verykios, "Duplicate Record Detection: A Survey," *IEEE Trans. Knowledge Data Eng.*, vol. 19, no. 1, pp. 1-16, Jan. 2007.

[2] E. Rahm and H.H. Do, "Data Cleaning: Problems and Current Approaches," *IEEE Data Eng. Bull.,* vol. 23, no. 4, pp. 3-13, Dec. 2000.
[3] A. Doan, Y. Lu, Y. Lee, and J. Han, "Object Matching for Information Integration: A Profiler-Based Approach," *IEEE Intelligent Systems,* vol. 18, no. 5, pp. 54-59, Sept. 2003.
[4] X. Dong, A. Halevy, and J. Madhavan, "Reference Reconciliation in Complex Information Spaces," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2005.
[5] M. Weis and F. Naumann, "Detecting Duplicates in Complex XML Data," *Proc. 22nd Int'l Conf. Data Eng. (ICDE),* 2006.
[6] I. Bhattacharya and L. Getoor, "Collective Entity Resolution in Relational Data," *ACM Trans. Knowledge Discovery from Data,* vol. 1, no. 1, pp. 1-36, Mar. 2007.
[7] M. Herschel and F. Naumann, "Scaling Up Duplicate Detection in Graph Data," *Proc. 17th ACM Conf. Information and Knowledge Management (CIKM) Conf.,* 2008.
[8] P. Singla and P. Domingos, "Object Identification with Attribute-Mediated Dependences," *Proc. European Conf. Principles and Practice of Knowledge Discovery in Databases (PKDD),* 2005.
[9] W. Shen, P. DeRose, L. Vu, A. Doan, and R. Ramakrishnan, "Source-Aware Entity Matching: A Compositional Approach," *Proc. IEEE 23rd Int'l Conf. Data Eng. (ICDE),* 2007.
[10] D.V. Kalashnikov and S. Mehrotra, "Domain-Independent Data Cleaning via Analysis of Entity-relationship Graph," *ACM Trans. Database Systems,* vol. 31, no. 2, pp. 716-767, 2006.
[11] Z. Chen, D.V. Kalashnikov, and S. Mehrotra, "Exploiting Relationships for Object Consolidation," *Proc. Second Int'l Workshop Information Quality in Information Systems (IQIS),* 2005.
[12] X. Yin, J. Han, and P.S. Yu, "LinkClus: Efficient Clustering via Heterogeneous Semantic Links," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB),* 2006.
[13] A. Arasu, C. Ré, and D. Suciu, "Large-Scale Deduplication with Constraints Using Dedupalog," *Proc. IEEE 25th Int'l Conf. Data Eng. (ICDE) Conf.,* 2009.
[14] M. Tachibana and H. Garcia-Molina, "Joint Entity Resolution," technical report, ID 900, Stanford InfoLab, 2009.
[15] R. Ananthakrishna, S. Chaudhuri, and V. Ganti, "Eliminating Fuzzy Duplicates n Data Warehouses," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB),* 2002.
[16] M.A. Hernández and S.J. Stolfo, "The Merge/purge Problem for Large Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 1995.
[17] S. Puhlmann, M. Weis, and F. Naumann, "XML Duplicate Detection Using Sorted Neigborhoods," *Proc. 10th Int'l Conf. Advances in Database Technology (EDBT),* 2006.
[18] M.J. Quinn and N. Deo, "Parallel Graph Algorithms," *ACM Computing Survey,* vol. 16, no. 3, pp. 319-348, 1984.
[19] H. sik Kim and D. Lee, "Parallel Linkage," *Proc. ACM Int'l Conf. Information and Knowledge Management (CIKM),* 2007.
[20] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proc. Sixth Conf. Symp. Operating Systems Design & Implementation (OSDI),* 2004.
[21] M. Weis and F. Naumann, "DogmatiX Tracks Down Duplicates in XML," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2005.
[22] I. Bhattacharya and L. Getoor, "Iterative Record Linkage for Cleaning and Integration," *Proc. Ninth ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery (DMKD),* 2004.
[23] A.E. Monge and C.P. Elkan, "An Efficient Domain-independent Algorithm for Detecting Approximately Duplicate Database Records," *Proc. SIGMOD Workshop Data Mining and Knowledge Discovery (DMKD),* 1997.
[24] M. Weis and F. Naumann, "Relationship-Based Duplicate Detection," Technical Report HU-IB-206, Humboldt Univ. Berlin, 2006.
[25] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster, "Industry-Scale Duplicate Detection," *Proc. Int'l Conf. Very Large Data Bases (VLDB),* 2008.
[26] M. Weis and F. Naumann, "Space and Time Scalability of Duplicate Detection in Graph Data," technical report, Nr. 25, Hasso-Plattner-Insitut Potsdam, 2008.
[27] D. Milano, M. Scannapieco, and T. Catarci, "Structure Aware XML Object Identification," *Proc. First Int'l Very Large Data Bases (VLDB) Workshop Clean Databases (CleanDB),* 2006.
[28] W.W. Cohen, P. Ravikumar, and S.E. Fienberg, "A Comparison of String Distance Metrics for Name-Matching Tasks," *Proc. IJCAI Workshop Information Integration on the Web (IIWeb),* pp. 73-78, 2003.
[29] B.-W. On, N. Koudas, D. Lee, and D. Srivastava, "Group Linkage," *Proc. IEEE 23rd Int'l Conf. Data Eng. (ICDE),* 2007.
[30] N. Reddy and J.R. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers," *Proc.31st Int'l Conf. Very Large Data Bases (VLDB),* 2005.

Melanie Herschel received the graduate degree in information technology from the University of Cooperative Education Stuttgart in 2003, and the PhD thesis on "XML duplicate detection" in 2007. She joined the information integration group at the Humboldt-University of Berlin (2003-2006) and continued her research on duplicate detection at the Hasso Plattner Institute in Potsdam (2006-2008). From 2008-2009, she worked at the IBM Almaden Research Center, focusing on data provenance. From 2009 to 2011, she was a postdoctoral researcher at the University of Tübingen, Germany. Since 2011, she has been an assistant professor at the University of Paris 11, France.

Felix Naumann studied mathematics, economy, and computer sciences at the University of Technology in Berlin. He received the diploma (MA) in 1997 and the PhD thesis on "quality-driven query answering" in 2000. After receiving the diploma (MA), he joined the graduate school "Distributed Information Systems" at Humboldt University of Berlin. In 2001 and 2002, he worked at the IBM Almaden Research Center on topics around data integration. From 2003-2006, he was assistant professor for information integration at the Humboldt-University of Berlin. Since 2006, he has been the chair for information systems at the Hasso Plattner Institute at the University of Potsdam in Germany.

Sascha Szott received the diploma in computer science from University Halle-Wittenberg with a thesis on XML schema matching. In 2007, he joined the Information Systems group at Hasso Plattner Institute, Potsdam, where he worked as a research assistant, focusing his research on data fusion and duplicate detection. Since 2009, he has been a member of the Scientific Information Systems group at Zuse Institute Berlin. In this position, he is responsible for the development of information systems, primarily for libraries and research institutions.

Maik Taubert studied software systems engineering at the Hasso Plattner Institute in Potsdam and received the master's degree in 2008 after finishing the master's thesis on parallelizing graph duplicate detection at the chair for information systems. Since then, he has been with Biotronik SE & Co. KG in Berlin developing the Home Monitoring Service Center for implantable devices.